Introduction to security and cryptography
Lecture 5: PKI and Key Management

Marius Lombard-Platet

ENS, Almerys

Today

PKI and Key Management

- PKI architecture
- How to identify someone, password storing

Public Key Infrastructure

How to talk with people

On the Internet, how do you know you are talking to the good person?

How to talk with people

On the Internet, how do you know you are talking to the good person?

You can if your messages are authenticated. Hence, you can if you use crypto!

We've seen two main modes of authentication in this class (can you remember which?)

How to talk with people

On the Internet, how do you know you are talking to the good person?

You can if your messages are authenticated. Hence, you can if you use crypto!

We've seen two main modes of authentication in this class (can you remember which?)

- MAC and HMAC
- Signatures

With MAC or symmetric crypto in general, you'll need to exchange keys for each pair of people. For 1000 people, that's 1000000 keys. With asymmetric crypto, it's only 1000.

Signatures are not enough

You know the signature is valid if it matches the public key of your correspondent.

How do you get your correspondent's key?

It works if you already know your correspondent's key in advance.

Signatures are not enough

You know the signature is valid if it matches the public key of your correspondent.

How do you get your correspondent's key?

It works if you already know your correspondent's key in advance.
But we can't do that for everyone.

Concept of Key Certificate

Main idea

- Alice trusts Bob and knows his public key
- Bob has signed asserting that Carol's key is K
- Then Alice may be willing to believe that Carol's key is K

Definition
A key certificate is an assertion that a certain key belongs to a certain entity, which is digitally signed by an entity (usually a different one).

Public Key Infrastructure (PKI)

Definition
PKI is an infrastructure build of certificates and servers to create, manage and publish certificate to allow autenticity certified by an authority.

Two Kinds of PKI

Hierarchical PKI

- Certificate Authorities are different of users
- X.509 (PKIX)

Non-Hierarchical PKI

- Each user manages his own trust network
- Pretty Good privacy (PGP)

Example `https` for Gmail

- Gmail sends to your browser its public key and a certificate signed by a certificate authority *Thawte Consulting (Pty) Ltd.* to prove that this key really is Gmail's key
- Your browser will verify Thawte's signature on Gmail's key using the public key of this reputable key certificate authority, stored in your browser
- Hence your browser trusts Gmail

Public Key Infrastructure (PKI)

PKI Mains Features

- Generation of public and private keys
- Certificate generation
- Giving the certificate to the owner
- Certificate publication
- Certificate verification
- Certificate revocation
- Others:
  - Protection of private key
  - Journalisation of actions
  - Revocations of private keys
  - Storage of certificates

CA: Certification Authority RA: Registration Authority

One issue with PKI

Main caveat: you have to trust the root CA.

Yet in several cases, Root CAs have forged rogue (malicious) certificates when governments asked them to.

One issue with PKI

Main caveat: you have to trust the root CA.

Yet in several cases, Root CAs have forged rogue (malicious) certificates when governments asked them to. For instance, CNNIC for Egypt in 2015, IGC/A by ANSSI (France), in 2013...

Sometimes, Root CAs also get hacked. (DigiNotar in 2013)

PGP: Pretty Good Privacy

Not a hierarchical PKI

I physically meet you, and I trust you. Therefore I sign your key, and you sign mine.

PGP: Pretty Good Privacy

Not a hierarchical PKI

I physically meet you, and I trust you. Therefore I sign your key, and you sign mine.

I will trust people that you trust as well

PGP: Pretty Good Privacy

Not a hierarchical PKI

I physically meet you, and I trust you. Therefore I sign your key, and you sign mine.

I will trust people that you trust as well, and the people trusted by the people you trust, and so on.

PGP: Pretty Good Privacy

Not a hierarchical PKI

I physically meet you, and I trust you. Therefore I sign your key, and you sign mine.

I will trust people that you trust as well, and the people trusted by the people you trust, and so on.This is called the **web of trust**.

PGP caveat

Being responsible and actually trusting people is difficult.

Why does no-one use PGP?

- It's not considered necessary.
- It's quite complicated. You need to spend a day to understand it properly. And even then, understanding is not guaranteed!
- It's a hassle. You need to maintain your keys, your web of trust, you need to configure your mail client.

Why Johnny can't encrypt is an article explaining why people can't/don't want to use PGP.

USER CONFIDENCE is among the most difficult things to achieve in computer security.

Identification

How to identify yourself?

- What you are: fingerprint, DNA, iris scan...
- What you know: password, the name of your first dog...
- What you have: your phone, a device thats displays a new password every minute...

What should we do?

Passwords are the most common used method, but also quite vulnerable. Passwords are leaked many times, and often predictable.

A good solution should mix at least two methods. For instance, the 2 Factor Authentication (2FA) asks for something you know (the password) and something you own (a code sent on your phone).

Passwords

Storing passwords in cleartext is a BAD IDEA: if the server is hacked, then everything is leaked.

Should we encrypt them?

Passwords

Storing passwords in cleartext is a BAD IDEA: if the server is hacked, then everything is leaked.

Should we encrypt them? No. Encryption means it can be decrypted, which is equally as bad.

The real solution is to use **hashing**: instead of storing a password, you store the hash of the password.

When someone enters their password, you hash it and verify the hash matches.

Passwords, again

Which kind of hash should you use? A cryptographic one of course. Recommended: BCrypt, Argon2[1].

Yet this is not enough! It is still easy to bruteforce small passwords.

Hence, a good application will choose one fixed random string *salt*, and store $H(password||salt)$. This way, bruteforcing passwords is much more difficult.

[1] It is most recommended that the hash function is *slow*, in order to avoid bruteforce attacks. Hence, SHA is not recommanded.

Crypto in Real Life

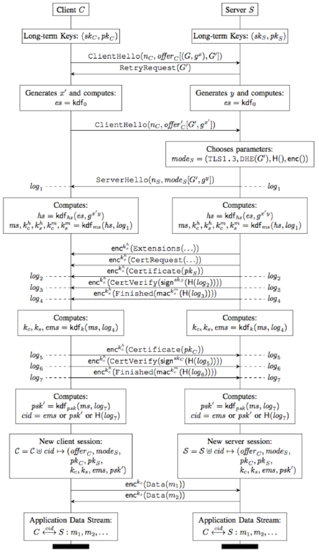
Example: TLS

SSL/TLS is the technology used to protect the web pages (HTTPS relies on SSL/TLS)

Goal: exchange an encryption key with the server, so the rest of the data can be safely encrypted.

Hence, we must guarantee integrity, authenticity and confidentiality.

In brief: TLS is complicated



Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^k(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) =$
$\quad \mathsf{HMAC\text{-}H}^s(len_{\mathsf{H}()} \| \text{“TLS 1.3,”} \| l \| h \| \mathtt{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where
$\quad ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$
$\quad hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1)$
$\quad hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$
$\quad k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \text{“”})$
$\quad k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \text{“”})$
$\quad k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \text{“”})$
$\quad k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where
$\quad ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4)$
$\quad ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4)$
$\quad ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \text{“”})$
$\quad k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where
$\quad psk' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where
$\quad es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk)$
$\quad k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where
$\quad ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \text{“”})$

TLS 1.3, quickly explained

1. First, the client contacts the server (`ClientHello`), and offers a list of cipher suites (ECDSA, ECDH, RSA...). They provide an ephemeral public key to the server
2. The server answers back with `ServerHello`, selects the cipher suite and returns their own ephemeral public key as well
3. From these exchanges, client and server derive a common secret key (`Handshake Keys`)
4. Now the channel is secured. The server encrypt their certificate and sends it to the user (`Server Certificate`)
5. The client acknowledges that the server is authenticated, then sends their own authentication to the server (`Client Certificate`, optional)
6. The server acknowledges, now both can exchange data as they want.

Crypto mechanisms in TLS

- TLS gives the choice between *several crypto mechanisms* so that each system, being different, will find what suits them
- Server *Authentication* (and client authentication, optionally) :
    - publick key : RSA, DSS, ECDSA
    - secret key or shared password : PSK (*Pre-Shared Key*), SRP (*Secure Remote Password*)
    - no authentication : ANON
- *Key exchange* :
    - public key (always the same one for the server) : RSA
    - Static Diffie-Hellman (same problem, always the same key) : DH, ECDH
    - secret key or shared password : PSK, SRP
    - Ephemeral Diffie Hellman (new keys for each connection) ; guarantees *forward secrecy*) : DHE, ECDHE

Crypto mechanisms in TLS

- *Encryption* :
  - block cipher : AES-CBC, 3DES-CBC, DES-CBC, etc.
  - block cipher with authentication mode : AES-CCM, AES-GCM, etc.
  - Flow encryption : RC4
  - No encryption : NULL (please don't do that)
- Message *Integrity* and *Origin authentication*:
  - HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
  - authentication mode for block cipher : AEAD
- A conbination of these mechanisms is called *cipher suite*
  - For instance : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- In TLS, data can also be compressed (zipped): NULL or DEFLATE

Some data about SSL/TLS

- more than 50 RFC
- 5 versions to this day
- more than 300 cipher suites
- more than 20 extensions
- other interesting functionalities: compression, renegociation...

Many implementations available.

Thank you for your attention

Any question?

What you should remember

- What is a PKI
- How does PGP work
- How does certification work
- How to store passwords
- 2FA
- Basic notion of how TLS works

Exercises

1. How should you react if one of your server gets hacked?
2. Two users have the same password. If the passwords are salted, will the two corresponding hashes be the same? Why?
3. What happens if you use an SSL certificate signed by you only (self-signed certificate)? Will the communication between you and the client be secure?